Chapter 16

Java Virtual Machine

To compile a java program in Simple.java, enter

```
javac Simple.java
```

javac outputs Simple.class, a file that contains *bytecode* (machine language for the *Java Virtual Machine* (JVM).

To run Simple.class, enter

```
java Simple
```

java (the Java interpreter) makes your computer act like the JVM so it can execute bytecode.

Why is Java slow?

- Interpretation of bytecode can involve a lot of overhead.
- JVM dynamically links classes.
- JVM performs checks during loading, linking, and executing bytecode.

Why is Java good for the Web?

- Bytecode is space efficient.
- Bytecode is portable to any system with a java interpreter.
- Java applets are safe to run.

Four parts of the JVM

- Execution engine (contains pc register)
- Method area (contains information on each class: bytecode, static variables, information needed for verification and linking).
- Java stack (the run time stack). Each *frame* of the Java stack contains a *local variable array* and an *operand stack*.
- heap (contains data associated with objects). Periodically, *garbage collection* deallocates objects in the heap that are no longer referenced.

There are two types of stacks in the JVM

- The *Java stack*
- The Java stack consists of frames, one frame for each method invocation. Each frame contains an *operand stack* and a local variable array.

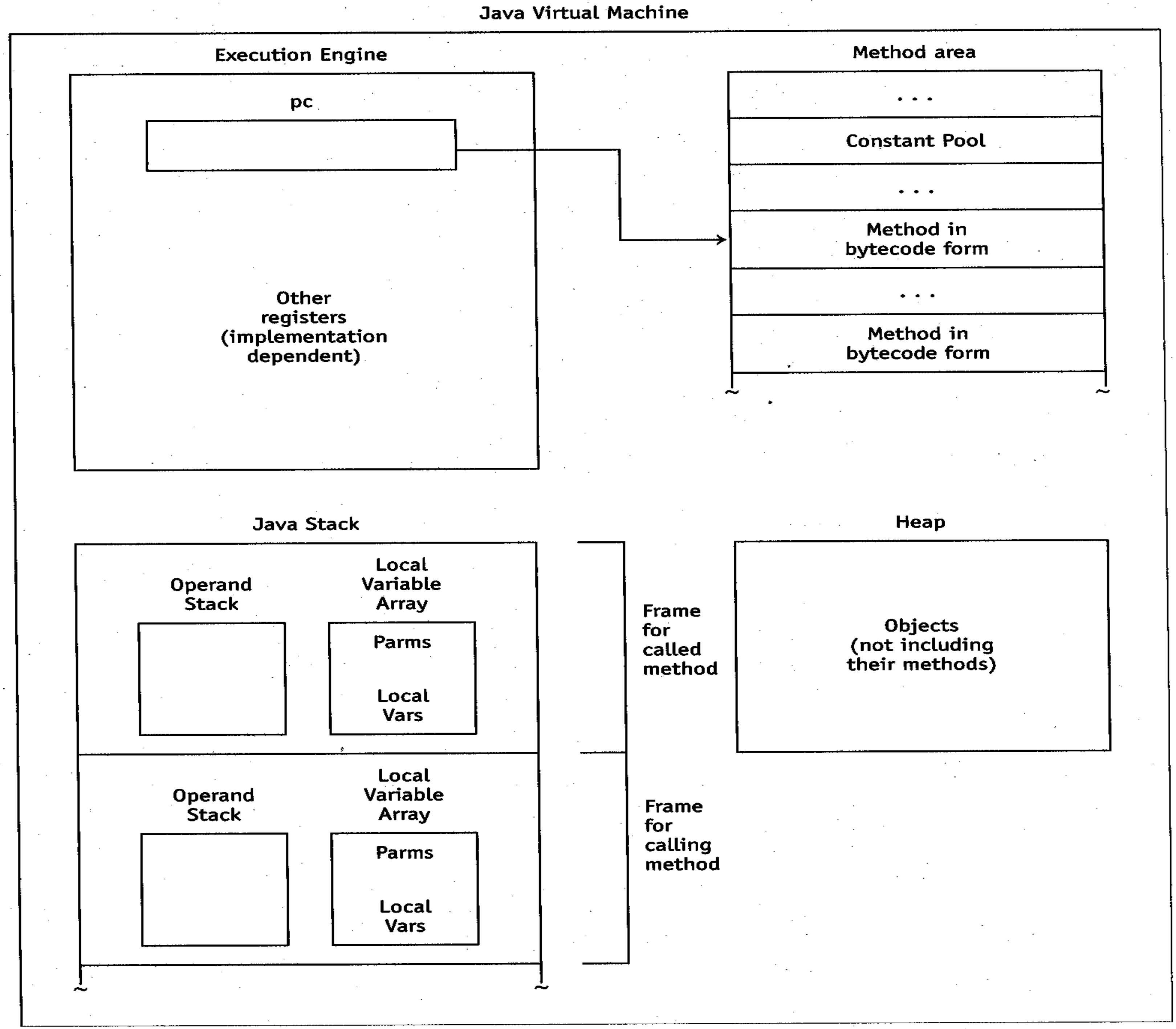
Local variable array

Contains local variables numbered starting from 0. For example, the first slot of the local variable array is called local variable 0.

Operand stack

Used to hold operands and results during the execution of instructions.

FIGURE 16.1



Java Virtual Machine
Execution Engine
pc
Other registers (implementation dependent)
Method area
. . .
Constant Pool
. . .
Method in bytecode form
. . .
Method in bytecode form
Java Stack
Operand Stack
Local Variable Array
Parms
Local Vars
Frame for called method
Operand Stack
Local Variable Array
Parms
Local Vars
Frame for calling method
Heap
Objects (not including their methods)

Some instructions consist of an opcode only. For example,

iconst_0, iconst_1, iconst_2,
iconst_3, iconst_4, iconst_5

which push 0, 1, 2, 3, 4, and 5, respectively, onto the operand stack.

The more common operations are performed by such opcode-only instructions.

Some instructions require an operand.
For example,

```
bipush 6
```

which pushes 6. This instruction consists of the opcode for bipush followed by a byte containing 6.

To push a number greater than 127, use sipush (short int push). For example,

```
sipush 130
```

Symbolic bytecode that adds three numbers

Let's say we wish to add 3, 6, and 130. One possible instruction sequence is

```
iconst_3            ; push 3
bipush    6         ; push 6
iadd                ; pop 6 and 3, add, and push sum (9)
sipush    130       ; push 130
iadd                ; add 130 and 9, add, and push sum
```

The initial letter of some mnemonics indicates the data type. For example, iadd, dadd, fadd, ladd.

a: reference

d: double

f: float

i: integer

ia: integer array

l: long

Load instructions on the JVM

iload_0

pushes the value in local variable 0 (i.e., it pushes the value from the first slot of the local variable array onto the operand stack.)

iload 4

pushes the value in local variable 4.

Store instructions on the JVM

istore_0

pops and stores the value on top of the operand stack into local variable 0.

istore   4

pops and stores the value on top of the operand stack into local variable 4.

A *static variable* in Java is a variable associated with a class rather than an object. It is shared by all objects of its class.

A *static method* in Java is a method that can be called via its class.

The getstatic and putstatic instructions transfer values between the top of the operand stack and static variables.

The operand that appears in getstatic and putstatic instructions is an index into the *constant pool*. For example,

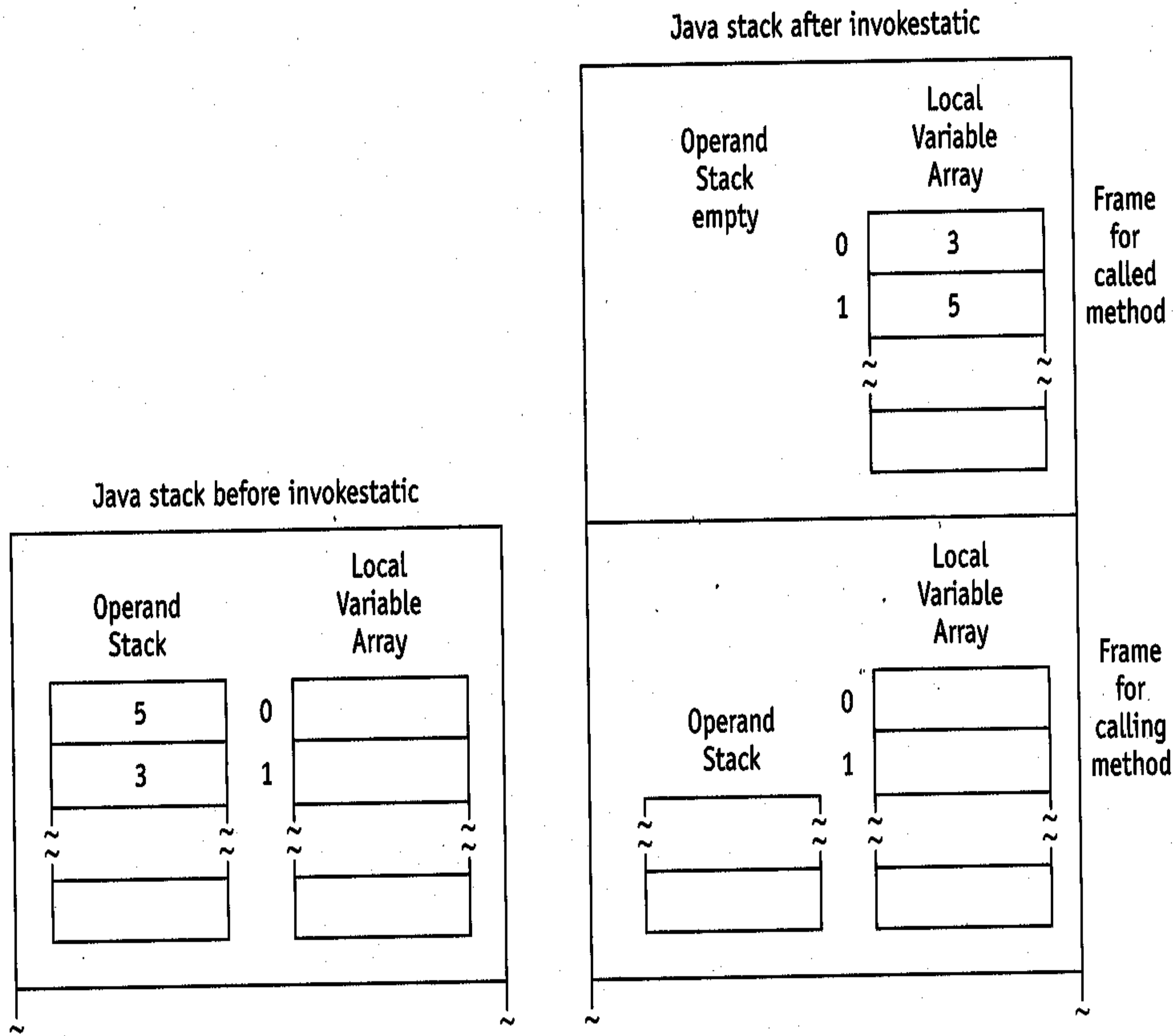
getstatic 2

2 is a constant pool index.

Invoking a static method with invokestatic instruction

- Creates frame for the called method and pushes it onto the Java stack.
- Pops the arguments from the caller's operand stack and places them in the called method's local variable array starting from local variable 0.
- Transfers control to the called method.

FIGURE 16.2

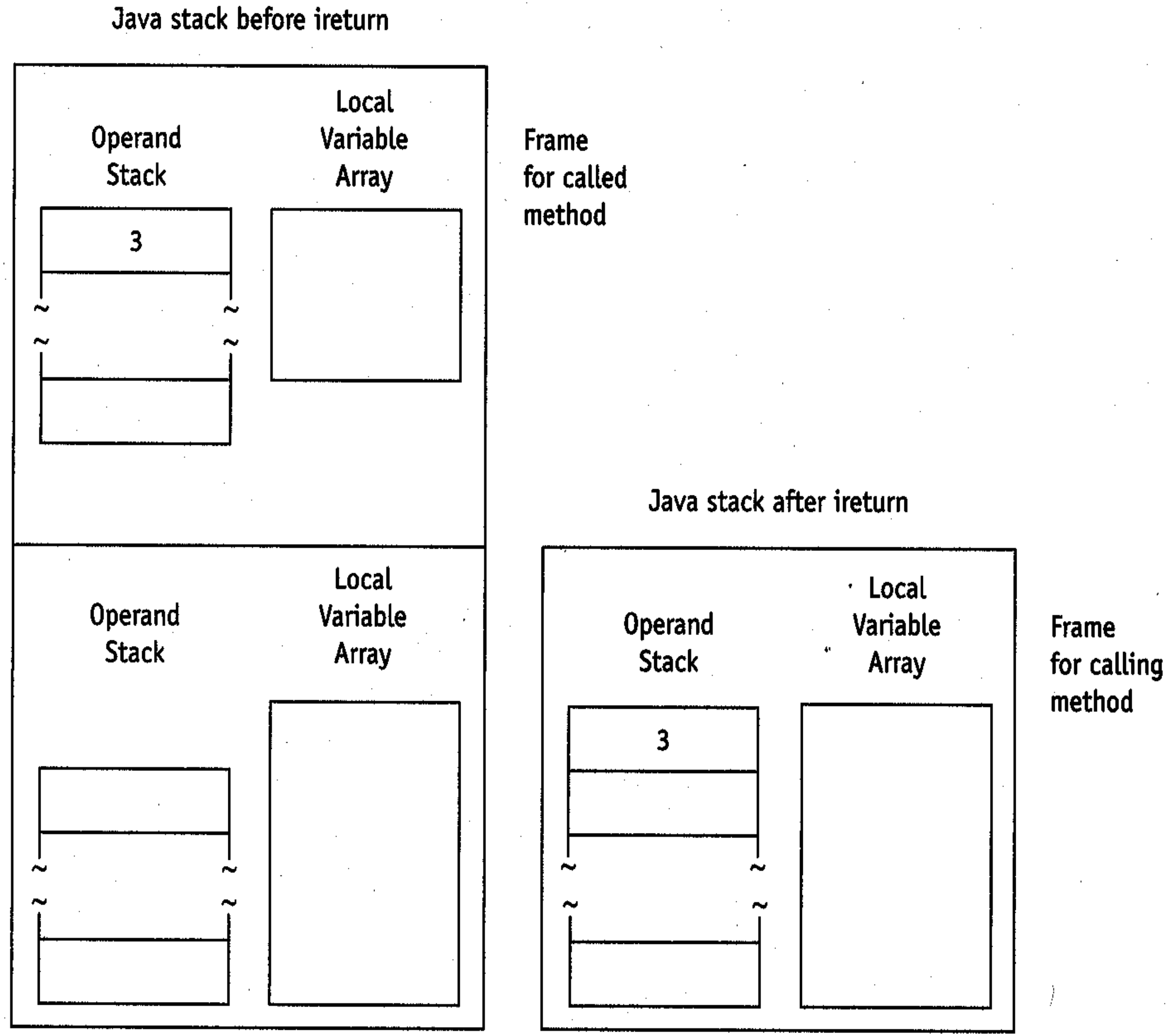

Java stack before invokestatic
Operand Stack
Local Variable Array
5
3
0
1
Java stack after invokestatic
Operand Stack empty
Local Variable Array
0
1
3
5
Frame for called method
Operand Stack
Local Variable Array
0
1
Frame for calling method

Returning a value to the calling method with the ireturn instruction

```
icont_3        ; push 3
ireturn        ; return 3
```

The value returned is pushed onto the calling method’s operand stack.

FIGURE 16.3


Java stack before ireturn
Operand Stack
Local Variable Array
3
Frame for called method
Operand Stack
Local Variable Array
Java stack after ireturn
Operand Stack
Local Variable Array
3
Frame for calling method

Implementation of the execution engine

The execution engine of the JVM repeatedly performs the four steps that a CPU typically performs:

1. Fetch the instruction.
2. Increment the pc.
3. Decode the opcode.
4. Execute the instruction.

FIGURE 16.4

```
Action                C++ Code
--------------------------------------------------------------------------
Fetch            | opcode =
                 |   (unsigned char)*pc;   // zero extend *pc
                 |
Increment pc     | pc++;                   // move pc over opcode
                 | switch
Decode           | [redacted] (opcode) {
Opcode           |
                 |   case 16:              // bipush opcode is 16
                 |     sp--;               // make room for push
                 |     sp[0] = (int)*pc;   // fetch and push operand
                 |     pc++;               // move pc over operand
                 |     break;
                 |
                 |   case 27:              // iload_1 opcode is 27
                 |     sp--;               // make room for push
                 |     sp[0] = ap[1];      // push loc var 1
                 |     break;
Execute          |
                 |   case 96:              // iadd opcode is 96
                 |     sp[1] =             // slot below top
                 |     sp[0] + sp[1];      // sum of top 2 slots
                 |     sp++;               // now pop top
                 |     break;
                 |
                 |   // other cases
                 | }
```

The wisdom of using a stack architecture for the JVM

- A stack architecture on a simulated machine is no slower than a register architecture.
- Bytecode is very compact which is important for a web programs.

A simple Java program follows, along with its bytecode

FIGURE 16.5

```
class Simple {
 static int gv1, gv2 = 5;

 // <init> method                   ; default constructor
 // 0  2A     aload_0                ; get object's reference
 // 1  B70001 invokespecial 1        ; invoke <init> in superclass
 // 4  B1     return
 //==============================================================
 public static void main(String arg[])
 {
   int lv1,
   lv2 = 7;       // 0  1007   bipush 7       ; push 7
                  // 2  3D     istore_2       ; store in lv2

   lv1 = 11;      // 3  100B   bipush 11      ; push 11
                  // 5  3C     istore_1       ; store in lv1

   gv1 =  fa(gv2, lv1, lv2);
                  // 6  B20002 getstatic 2    ; push gv2
                  // 9  1B     iload_1        ; push lv1
                  // 10 1C     iload_2        ; push lv2
                  // 11 B80003 invokestatic 3 ; call fa
                  // 14 B30004 putstatic 4    ; pop into gv1
                  // 17 B1     return
 }
```

(continued)

FIGURE 16.5 (continued)

```
//=================================================================
public static int fa(int x, int y, int z)
{
  return x + y + z; // 0  1A  iload_0        ; push x
                    // 1  1B  iload_1        ; push y
                    // 2  60  iadd           ; pop/pop/add/push
                    // 3  1C  iload_2        ; push z
                    // 4  60  iadd           ; pop/pop/add/push
                    // 5  AC  ireturn        ; pop and return
}
//=================================================================
// <clinit> method                 ; class initializer
// 0  08     iconst_5              ; push 5
// 1  B30002 putstatic 2           ; pop into gv2
// 4  B1     return
}
```

A formatted display of the constant pool for our simple program follows.

FIGURE 16.6

3 (operand in the invokestatic instruction on line 22 in Fig. 16.5)

Constant Pool

Index	Tag		Information			
1:	10	(Methodref)	6	(Class Index);	21	(NameAndType Index)
2:	9	(Fieldref)	5	(Class Index);	22	(NameAndType Index)
3:	10	(Methodref)	5	(Class Index);	23	(NameAndType Index)
4:	9	(Fieldref)	5	(Class Index);	24	(NameAndType Index)
5:	7	(Class)	25			
6:	7	(Class)	26			
7:	1	(UTF8)	"gv1"			
8:	1	(UTF8)	"I"			
9:	1	(UTF8)	"gv2"			
10:	1	(UTF8)	"<init>"			
11:	1	(UTF8)	"()V"			
12:	1	(UTF8)	"Code"			
13:	1	(UTF8)	"LineNumberTable"			
14:	1	(UTF8)	"main"			
15:	1	(UTF8)	"([Ljava/lang/string;)V"			
16:	1	(UTF8)	"fa"			
17:	1	(UTF8)	"(III)I"			
18:	1	(UTF8)	"<clinit>"			
19:	1	(UTF8)	"SourceFile"			
20:	1	(UTF8)	"Simple.java"			
21:	12	(NameAndType)	10	(Name Index);	11	(Descriptor Index)
22:	12	(NameAndType)	9	(Name Index);	8	(Descriptor Index)
23:	12	(NameAndType)	16	(Name Index);	17	(Descriptor Index)
24:	12	(NameAndType)	7	(Name Index);	8	(Descriptor Index)
25:	1	(UTF8)	"Simple"			
26:	1	(UTF8)	"java/lang/Object"			

Information in the constant pool for index 3

```
3 -> 5 -> 25 "Simple"
```

which yields `"Simple"`, the class name of the method. Similarly, the chains

```
3 -> 23 -> 16 "fa"
3 -> 23 -> 17 "(III)I"
```

An attribute in a class file

```
0013        Attribute name index ("SourceFile")
00000002    Attribute length (length of what follows this field)
0014        Name index ("Simple.java")
```

The first entry is the constant pool index of the attribute name. The second entry is the length of what follows.

A hex display of the complete class file for our simple program follows.

FIGURE 16.7

Hex Display of Simple.class

```
          CAFE BABE  Magic number (signature for class files)
          0003       Minor version number of JVM
          002D       Major version number of JVM
-------------------------------------------------------------
Constant pool

          001B       Constant pool count
Index     Tag
 01:      0A   0006 0015
 02:      09   0005 0016
 03:      0A   0005 0017
 04:      09   0005 0018
 05:      07   0019
 06:      07   001A
 07:      01   0003 6776 31                        "gv1"
 08:      01   0001 49                             "I"
 09:      01   0003 6776 32                        "gv2"
 0A:      01   0006 3C69 6E69 743E                 "<init>"
 0B:      01   0003 2829 56                        "()V"
```

(continued)

FIGURE 16.7 (continued)

```
0C:    01   0004 436F 6465                                        "Code"
0D:    01   000F 4C69 6E65 4E75 6D62 6572 5461 626C 65
                                                                  "LineNumberTable"
0E:    01   0004 6D61 696E                                        "main"
0F:    01   0016 285B 4C6A 6176 612F 6C61 6E67 2F53 7472 696E 673B 2956
                                                                  "([Ljava/lang/String;)V"
10:    01   0002 6661                                             "fa"
11:    01   0006 2849 4949 2949                                   "(III)I"
12:    01   0008 3C63 6C69 6E69 743E                              "<clinit>"
13:    01   000A 536F 7572 6365 4669 6C65                         "SourceFile"
14:    01   000B 53 696D 706C 652E 6A61 7661                      "Simple.java"
15:    0C   000A 000B
16:    0C   0009 0008
17:    0C   0010 0011
18:    0C   0007 0008
19:    01   0006 5369 6D70 6C65                                   "Simple"
1A:    01   0010 6A61 7661 2F6C 616E 672F 4F62 6A65 6374
                                                                  "java/lang/Object"
-------------------------------------------------------------------------------
0020         Access flags (0020 indicates that JVM should
               use newer version of invokespecial instruction)
0005         This class index ("Simple")
0006         Super class index ("java/lang/Object")
0000         Interfaces count
0002         Fields count
-------------------------------------------------------------------------------
gv1
 0008        Access flags (0008 indicates static)
 0007        Name index ("gv1")
 0008        Descriptor index ("I")
 0000        Attributes count
-------------------------------------------------------------------------------
gv2
 0008        Access flags (0008 indicates static)
 0009        Name index ("gv2")
 0008        Descriptor index ("I")
 0000        Attributes count
-------------------------------------------------------------------------------
 0004        Number of methods
-------------------------------------------------------------------------------
<init>
 0000        Access flags (0000 indicates default access)
 000A        Name index of method ("<init>")
 000B        Descriptor index     ("()V")
 0001        Attributes count
```

(continued)

FIGURE 16.7 (continued)

```
 000C          Attribute name index ("Code")                     ┐
 0000001D      Attribute length                                  │
 0001          Max stack                                         │
 0001          Max locals                                        │
 00000005      Code length                                       │
 2A B70001 B1    Bytecode                                        │ Code
 0000          Exceptions count                                  │ Attribute
 0001          Attributes count                                  │
                                                                 │
 000D          Attribute Name Index ("LineNumberTable")  ┐ Line  │
 00000006      Attribute length                          │ Number│
 0001          LineNumberTable length                    │ Table │
 0000 0001     Location/line number                      ┘ Attribute
                                                                 ┘
----------------------------------------------------------------
main
 0009          Access flags (0009 indicates public and static)
 000E          Name index of method ("main")
 000F          Descriptor index ("([Ljava/lang/String;)V")
 0001          Attribute count

 000C          Attribute name index ("Code")                     ┐
 00000036      Attribute length                                  │
 0003          Max stack                                         │
 0003          Max locals                                        │
 00000012      Code length                                       │
 1007 3D 100B 3C B20002 1B 1C B80003 B30004 B1   Bytecode        │ Code
 0000          Exceptions count                                  │ Attribute
 0001          Attribute count                                   │
                                                                 │
 000D          Attribute name index ("LineNumberTable")  ┐ Line  │
 00000012      Attribute length                          │ Number│
 0004          LineNumberTable length                    │ Table │
 0000 000C     Location/line number                      │ Attribute
 0003 000F     Location/line number                      │       │
 0006 0012     Location/line number                      │       │
 0011 0019     Location/line number                      ┘       ┘

----------------------------------------------------------------
fa
 0009          Access flags (0009 indicates public and static)
 0010          Name index of method ("fa")
 0011          Descriptor index ("(III)I")
 0001          Attribute count
```

(continued)

FIGURE 16.7 (continued)

```

000C          Attribute name index ("Code")
0000001E      Attribute length
0002          Max stack
0003          Max locals
00000006      Code length
1A 1B 60 1C 60 AC    Bytecode                                          Code
0000          Exceptions count                                         Attribute
0001          Attribute count

000D          Attribute name index ("LineNumberTable")   Line
00000006      Attribute length                           Number
0001          LineNumberTable length                     Table
0000 001D     Location/line number                       Attribute

-------------------------------------------------------------
<clinit>
0008          Access flags (0008 indicates static)
0012          Name index of method ("<clinit>")
000B          Descriptor index ("()V")
0001          Attribute count

000C          Attribute name index ("Code")
0000001D      Attribute length
0001          Max stack
0000          Max locals
00000005      Code length
08 B30002 B1        Bytecode                                           Code
0000          Exceptions count                                         Attribute
0001          Attribute count

000D          Attribute name index ("LineNumberTable")   Line
00000006      Attribute length                           Number
0001          LineNumberTable length                     Table
0000 0002     Location/line number                       Attribute

-------------------------------------------------------------
0001           Attributes count

0013           Attribute name index ("SourceFile")   SourceFile
00000002       Attribute length                      Attribute
0014           Name index ("Simple.java")
```

Sizes of comparable programs

FIGURE 16.8

	SPARC	Pentium	Bytecode
main	40	52	18
fa	12	14	6
<init>	-	-	5
<clinit>	-	-	5
Total bytes	52	66	34

Some comparison and control instructions

- goto unconditional jump
- if_cmplt compares top two stack items
- if_icmpge compares top two stack items
- iflt compares top of stack with 0
- if_acmpeq compares references
- if_acmpne compares references

  See the illustrative program on the next slide.

FIGURE 16.9

```
class Control {
 public static void main(String arg[])
 {
     int x = 3,          // 0   06          iconst_3   ; push 3
                         // 1   3C          istore_1   ; pop into x

         y = 4;          // 2   07          iconst_4   ; push 4
                         // 3   3D          istore_2   ; pop into y

     while (x < 10 )     // 4   A7 0006     goto 6     ; goto loc 4+6

        x++;             // 7   84 01 01    iinc 1 1   ; add 1 to 1

     // (x < 10)            10  1B          iload_1    ; push x
     // exit test           11  10 0A       bipush 10  ; push 10
     // is here             13  A1 FFFA     if_icmplt -6 ; goto loc 13-6

     if (x < y)          // 16  1B          iload_1     ; push x
                         // 17  1C          iload_2     ; push y
                         // 18  A2 0009     if_icmpge 9 ; goto 18+9

        x = 20;          // 21  10 14       bipush 20   ; push 20
                         // 23  3C          istore_1    ; store in x

     else                // 24  A7 0006     goto 6      ; goto 24+6
```

(continued)

FIGURE 16.9 (continued)

```

        x = 30;        // 27 10 1E    bipush 30   ; push 30
                       // 29 3C       istore_1    ; pop into x

                       // 30 B1       return      ; return to caller
    }
}
```

Instructions that jump use pc-relative addressing

A70006 (the machine code for goto 6) jumps to the location whose address is 6 + the contents of the pc register (before incrementation).

Unassembling the Simple class file

javap –c Simple

FIGURE 16.10

```
Compiled from Simple.java
class Simple extends java.lang.Object {
   static int gv1;
   static int gv2;
   Simple();
   public static void main(java.lang.String[]);
   public static int fa(int, int, int);
   static {};
}

 Method Simple()          ← <init> method
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

 Method void main(java.lang.String[])
   0 bipush 7              constant pool index
   2 istore_2                     symbolic info obtained via constant
   3 bipush 11                                                pool index 2
   5 istore_1
   6 getstatic #2 <Field int gv2>
   9 iload_1
   10 iload_2
   11 invokestatic #3 <Method int fa(int, int, int)>
   14 putstatic #4 <Field int gv1>
   17 return

 Method int fa(int, int, int)
   0 iload_0
   1 iload_1
   2 iadd
   3 iload_2
   4 iadd
   5 ireturn

 Method static {}         ←<clinit> method
   0 iconst_5
   1 putstatic #2 <Field int gv2>
   4 return
```

Unassemble this program to see its bytecode

FIGURE 16.11 a)

```
class IRTest {
    public static void main(String arg[])
    {
        int y;
        y = sum(1, 2);
    }
    static int sum(int m, int n)
    {
        return m + n;
    }
}
```

b)

```
Method void main(java.lang.String[])
    0 iconst_1              ; push 1
    1 iconst_2              ; push 2
    2 invokestatic #2 <Method int sum(int, int)>
    5 istore_1              ; save value returned on stack in loc var 1
    6 return

Method int sum(int, int)
                            ; return m + n;
    0 iload_0               ; get 1st parameter from loc var 0
    1 iload_1               ; get 2nd parameter from loc var 1
    2 iadd                  ; pop/pop/add/push
    3 ireturn               ; return value on top of stack
```

Arrays and objects

FIGURE 16.12 a)

```
class OATest {
 public static void main(String arg[])
 {
     O o;
     int a[];
     o = new O();
     o.x = 6;
     o.f();
     a = new int[7];
     a[5] = 3;
 }
}
```

b)

```
Method void main(java.lang.String[])
    0 new #2 <Class O>                          ;  o = new O();
    3 dup                                             ; duplicate reference
    4 invokespecial #3 <Method O()>
    7 astore_1                                        ; save reference

    8 aload_1                                    ;  o.x = 6;
    9 bipush 6
   11 putfield #4 <Field int x>

   14 aload_1                                    ;  o.f();
   15 invokevirtual #5 <Method void f()>

   18 bipush 7                                   ;  a = new int[7];
   20 newarray int
   22 astore_2                                        ; save reference

   23 aload_2                                    ;  a[5] = 3;
   24 iconst_5
   25 iconst_3
   26 iastore

   27 return
```

Now that you know the basics of the JVM, you can enjoy (and understand) some more advanced discussions of the JVM.

FURTHER READING

Engel, J. *Programming for the Java Virtual Machine*. Reading, MA: Addison-Wesley, 1999.

Harold, E. *Java Secrets*. Forster City, CA: IDG Books Worldwide, 1997.

Lindholm, T. and Yellin, F. *The Java Virtual Machine Specification*. Reading, MA: Addison-Wesley, 1997.

Meyer, J. and Downing, T. *Java Virtual Machine*. Sebastopol, CA: O'Reilly, 1997.

Venners, B. *Inside the Java Virtual Machine*. New York: McGraw-Hill, 1998.

return 0;